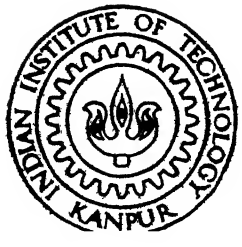
Design and Implementation of a File System with on-the-fly Data Compression for Unix

by

Praveen B



Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KANP

July, 1997

Design and Implementation of a File System with on-the-fly Data Compression For UNIX

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Praveen B

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

July. 1997

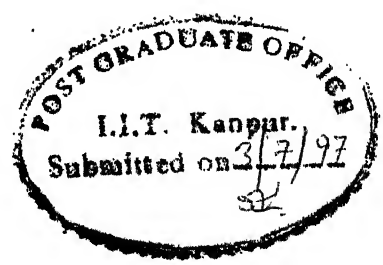
CSE-1997-M-PRA-DES

CERTIFICATE

This is to certify that the work contained in the thesis titled Design and Implementation of a File System with on-the-fly Data Compression for UNIX by Praveen B has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Deepak Gupta,
Assistant Professor,
Dept. of CSE,
IIT Kanpur.

Dr. Rajat Moona,
Associate Professor,
Dept. of CSE,
IIT Kanpur.



Acknowledgements

I am greatly indebted to my thesis supervisors, Dr.Deepak Gupta and Dr.Rajat Moona for their constant support and valuable guidance through out my work. Sirs, I am grateful to you.

Dr.Ajai Jain and Dr.Harish Parthasarthy have kindly agreed to be the examiners for my defence and have made their valuable comments and suggestions on my report. Sirs, I am thankful to you.

The world would have been a dull place to live but for the friends, who share and care every thought of ours. My friends Rama Krishna Rao and K.Srinivas have stood by me in all the times. Friendly moments spent with Shyamala and Malini have always made me recollect the playful days of my childhood and have been a source of immense joy and fun and remain forever as sweet memories. Associations with Ramakanth, Raghu Ram, Sameer Shah etal., stand as cherished times. Two friends, Jayaram and Vani, though thousands of miles away, have provided me the right guidance and emotional support in times of need. Friends, I adore you all.

It is the constant love and care of my parents, my brother and my cute sister that made me, what I today am. Dear Amma, Daddy, Kinnu and Deepa, I love you all.

The cozy surroundings of iitk has made my stay here, certainly one of the core modules of this package of life. Dear birds, trees and flowers of this beautiful campus, I cherish your company forever.

Dedicated to

A Good Kind Honest and Simple Friend

Abstract

Data compression techniques have long been assisting in making effective use of disk, network and similar resources. *Compress* for UNIX, *pkzip* for MS-DOS, *gzip* developed by GNU software foundation are a few widely used utilities for this purpose. These utilities require explicit user interaction for compressing and uncompressing of file data. *Doublespace* for MS-DOS, *Stacker* for MS-DOS and Macintosh are utilities in which compression and uncompression of file data takes place in the background, without any user interaction. A compressed file requires less number of sectors for storage on the disk. Hence, incorporating data compression techniques into a file system gives the advantage of larger effective disk space. At the same time, the time lost in compression and uncompression of file data gets compensated to some extent by the time gained because of lesser disk access. In this report we describe the design and implementation of a file system, with the feature of *on-the-fly* data compression in a fashion that is transparent to the user, for Linux, a Unix-like operating system. We also present some experimental results that show that the performance of our file system is comparable to that of Ext2fs, the native file system for Linux.

Contents

Chapter 1

Introduction

Data compression techniques have long been assisting in making effective use of disk, network and similar resources. Compressing the data before storing it on the disk or transferring it over a network reduces the amount of bandwidth required for the data transfer. Data compression when used in the context of a file system stored on a physical disk has the following advantages.

Better Disk Utilization

When a file is compressed before being written onto the disk, in most of the cases it occupies less number of disk sectors than the corresponding uncompressed file. This leads to efficient usage of the available disk space in the sense that the same amount of disk space can now be used to store a larger amount of data. This advantage is however dependent on the amount of compression that is achieved.

In certain file systems like the Ext2 file system[2] developed for Linux[8], a file system partition is divided into multiple block groups. Block allocation strategies always try to allocate the data blocks for a file in the same group as its inode. This reduces the disk seeks, when a process tries to access an inode and its data blocks. Since a file in the compressed form requires less number of disk blocks for storage, the inode and the data blocks for files in the file system can now be more effectively grouped.

Faster Disk Access

A considerable amount of time of the *read* and *write* system calls is spent in transferring data from and to the disk. With the advent of faster CPUs, the ratio of CPU time to the disk access time for reading and writing files is decreasing fast.

Since storing a file in the compressed form on the disk requires lesser number of disk blocks, disk access will take lesser time if the file is compressed, since we need to access lesser number of sectors on the disk. Thus by choosing a compression algorithm that is fairly fast, the time lost in compressing and uncompressing the data can be compensated for by the time gained in disk access. This leads to the advantage of gaining in terms of disk space with not much loss in terms of performance of the file system.

1.1 Related Work

1.1.1 Compression Utilities

A large number of utilities have been developed in assisting the users store their data in the compressed form. *Compress* and *pack* for Unix; *pkzip* for MS-DOS; and *gzip* developed by the GNU software foundation are a few of the widely used compression utilities. Most of these compression routines use variations of the dictionary based compression technique proposed by Lampel-Ziv[11]. Using these utilities for data compression imply that the user explicitly needs to compress the data before storage, and do the uncompression when needed.

1.1.2 Stacker

Stacker is a program developed for MS-DOS which doubles the disk space on the drive invisibly[10]. It allows more data to be placed on the disk by compressing all the files on the disk. It uses a method called LZS compression, which replaces repeating strings of characters with a 'token' character. Each token represents a different string of characters. As data is read from the disk, tokens are replaced

with the original characters. This method is a form of lossless compression. This means that Stacker will never lose or misplace the original information.

Stacker runs in the background of the operating system. It is invisible to applications, disk utilities, and even the operating system. Stacker does this by working hand-in-hand with the hard disk driver. The hard disk driver is the software that communicates between the hard disk and the operating system. Without a hard disk driver, the hard disk would not be accessible. Stacker does not change the hard disk driver in any way.

The Stacker driver operates between the operating system and the hard disk driver. When the operating system wants to interact with the drive, it calls Stacker. As far as the operating system is concerned, Stacker is the hard disk driver. Stacker performs the compression or uncompression as needed and then passes the information on to the hard disk driver. The hard disk driver does not care what the information looks like. It just reads and writes the data to the disk as instructed.

This mode of operation allows Stacker to work invisibly in the background. The files keep the same attributes. They look as they did before Stacker was installed. The files appear to be of the same size, and as far as the operating system is concerned, they are.

Stacker tracks all of the information in the background using *compression tables* it adds to the system. If for any reason the compression tables should become corrupted, Stacker includes a Check feature. Using the check feature, Stacker can repair the compression table information. Stacker maintains redundant copies of these tables so as to be able to better recover from problems.

1.1.3 Doublespace

Doublespace[5] is a disk compression utility which is included in MS-DOS 6.0. Its function and operation are similar to most other disk compression utilities like Stacker, SuperStor[6][7] etc. Its main purpose is to allow more information to be stored on the diskette. This process occurs at the system level and is for the most part, transparent to the user.

When Doublespace is run on a standard drive like the C: drive, a new drive will appear on the user system. This new drive will be uncompressed and will be assigned a drive letter of H: (default). This drive will contain about 2MB of free space, system files and Doublespace files. The free space on this drive will be allocated for files that do not function properly when they are compressed. One file DBLSPACE.000, is a CVF (compressed volume file) and represents the entire compressed C: drive. This file is in a compressed form and resides in the root directory of the host drive. Doublespace automatically uncompresses the files in the compressed drive before they are used.

1.2 Organization of the Report

In this report, we present the design and implementation of a compressed file system for a Unix-like operating system. We start with the background concepts of the Unix file system and a brief introduction to the data compression techniques in Chapter 2. In Chapter 3 we present the design of our file system. Chapter 4 contains details of implementation of our file system on Linux[8], a Unix-like operating system. In Chapter 5 we present the results of benchmarking the performance of our implementation. Finally in Chapter 6, we present the concluding remarks.

Chapter 2

Background Concepts

2.1 The Unix File System

2.1.1 Structure

A file in a Unix operating system[1] is a sequence of bytes with no interpretation enforced on it by the operating system. The file system is organized as a tree of files in which the leaf nodes are the regular files and the intermediate nodes are directories. A directory in a Unix file system is a special file which contains a list of entries, one entry each for the sub-directories and the files in it.

- *Inodes*

Each file is represented by a structure, called an *inode*. An inode contains all the necessary information about the file it represents, such as the file type, size, owner, access rights, time stamps, and pointers to the data blocks. The addresses of data blocks allocated to the file are stored in its inode. When a user requests an I/O operation on the file, the kernel code converts the current offset in the file to the logical block number within the file, uses this number as the index into the block entry table and reads or writes the corresponding physical block(s).

- *Directories*

Directories in Unix are just like other files except that the information they contain is in a specific format and has a specific interpretation enforced by the operating system. Each directory contains one entry for each of the sub-directories and files in it. Each entry contains the inode number and the name of the file.

- *Links*

Unix file system implements the concept of a link, whereby each file can have multiple names associated with it. This is implemented by associating various file names with the same inode. The inode has a count of number of links pointing to it. When a link is deleted, the kernel simply removes the corresponding entry from the directory and decrements the link count in the inode. An inode and its data blocks are deallocated only when its link count becomes zero.

2.1.2 Buffer Cache

In a Unix file system, when a process tries to access data from a file, the operating system brings data from the disk into a buffer and gives a copy of it to the requesting process. Future accesses to the same data are satisfied by using this cached copy itself. The kernel could read and write directly from the disk for each request, but that would severely affect the system response and throughput because of slower disk access rates. The kernel therefore tries to minimize the frequency of disk accesses by maintaining a pool of internal data buffers called the buffer cache, which contains data of the recently accessed disk blocks.

On getting a read request for a file, the kernel attempts to find the data in the buffer cache. If the data is already present there, disk access is not necessary for satisfying the read request. If the data is not present, the kernel reads the data from the disk into the buffer cache which may be used for future accesses. Data being written is also cached so that it is available if any other process later tries to read it. The kernel also tries to minimize disk writes by not writing back any transient data. Pre-caching and delayed write of data also aid in minimizing the disk accesses.

The buffer cache is organized as a hash queue, hashed upon the device and block number on the disk whose data is stored in the corresponding buffer. A buffer can exist on only one hash queue, since otherwise the kernel has no way to decide which of them is the valid buffer for that <device,block> pair. The size of this hash queue varies during the lifetime of the system, depending upon the number of buffer cache entries that are currently mapped to the disk blocks.

2.1.3 The Virtual File System

Most of the UNIX kernels contain a Virtual File System (VFS) layer similar to the one that was originally introduced in SunOS for supporting NFS[12]. The VFS is an indirection layer which handles the file oriented system calls, and calls the necessary functions in the file system specific code to do the input-output.

When a process issues a file system related system call, the kernel calls a function in the VFS code. This function handles the structure independent manipulations and redirects the call to a function contained in the file system specific code, which is responsible for handling the structure dependent operations. The file system specific code uses the buffer cache functions to request I/O on devices.

2.2 Data Compression Techniques

Data compression techniques[11] are broadly classified into

- **Statistical Models**

  Statistical modeling reads in and encodes a single symbol at a time using the probability of that character's appearance. The simplest forms of statistical modeling use a static table of probabilities for each symbol in the alphabet. The symbol that occurs with high probability is encoded with less number of bits and vice versa.

  However, using static tables of probabilities has a disadvantage. If the input stream does not match well with the table of probabilities, it leads to poor

compression. Hence another technique called the Adaptive Statistical Modeling is used, whereby the probability tables are generated as the input is scanned and compressed. Thus, the algorithm may start with a small table or no table at all, and as the input is scanned, the probability tables are built up.

An implementation of the Adaptive Statistical Modeling technique is the Huffman Encoding scheme, where a Huffman tree is generated for the input symbols seen. This tree is used for encoding the input symbols.

- **Dictionary Schemes**

  Statistical models generally encode a single symbol at a time. A dictionary based compression scheme uses a different concept. It reads in input data and looks for a group of symbols that appear in the dictionary. If a string match is found, a pointer or index into the dictionary can be output instead of the code for the symbol. The longer the match, the better the compression.

  Dictionary based compression techniques differ in the way they maintain the dictionary, and the way they encode the dictionary match. The 12-bit LZW compression technique is an implementation of the dictionary based compression technique. It uses 12 bits to encode the dictionary match. The dictionary is maintained as a tree. Another variation of the dictionary based technique is the LZRW1 compression technique [14] which uses 16 bits to encode the dictionary match. The dictionary in this case is maintained as a hash table.

Chapter 3

Design of the Compressed File System

3.1 Motivation

The compressed file system has been designed with the aim of incorporating the feature of *on-the-fly* compression and uncompression of file data into the Unix-like file system, in a fashion that is transparent to the user. Our goal is to design a file system which makes effective use of the available disk space, by compressing the file before being written onto the disk, and then uncompressing it before returning to the user without much loss in the performance of the file operations.

The design has been guided by the idea that it should be possible to integrate our file system into a Unix-like operating system which provides support for multiple file systems, with minor modifications to the generic file system code.

We aim at providing this feature of *on-the-fly* compression and uncompression in a fashion that is transparent to the user. The user should be unaware of the fact that the file is stored in a compressed form on the disk. The size and other attributes of the file as seen by him will be the same as that of the uncompressed file.

However, we restrict the compression only to regular files. Since directories are usually smaller, storing them in compressed form does not give any advantage. On

the other hand, since they are more frequently used, compressing them affects the speed of the directory operations, thus affecting the file system performance. Hence we keep all the directories in the uncompressed form. Also, since compressing very small files is not desirable, we adopt a strategy to store all the files whose sizes are below a threshold value in the uncompressed form. Other special files such as symbolic links, pipes etc., are also not compressed.

Another aim of our design is to allow multiple compression techniques to be simultaneously used in the file system. This feature provides the option of selecting a suitable compression algorithm for a specific file at the time of creating that file.

3.2 Concepts of Physical and Logical Block Sizes

3.2.1 Physical Block Size

The physical block size of a file system indicates the smallest units into which the disk partition for this file system is divided. It is configured at the time of creating the file system on a certain disk partition, and is a constant for the file system in question. Physical block size of a file system will always be an integral multiple of the sector size of the disk. Any allocations of disk space for the files of a file system are always done in multiples of the physical block size of that file system.

3.2.2 Logical Block Size

The logical block size of a file system is a parameter that determines in what units are file reads and writes performed on that file system. In a typical Unix- like file system, the disk blocks allocated to a file are stored in the block entry table of the inode. The index of this table represents the logical block of the file and the corresponding table entry represents the physical blocks allocated to that logical block. The representation of the block entry table and its entries is specific to the implementation of the file system. The number of disk access requests required to read or write a file varies inversely with the size of the logical block, which thus affects the performance of the file system. However selecting a larger logical block

size may lead to internal fragmentation of the last logical block of the file. In most of the cases, the logical block size of a file system is equal to its physical block size. The logical block size of the file system is chosen at the time of creating the file system.

3.2.3 Need for different physical and logical block sizes

In order to store a file in the compressed form, we need to divide the file data into chunks of fixed size and each chunk is compressed and uncompressed separately. The whole file cannot be compressed as a single chunk as it leads to a heavy degradation of performance while reading and writing small parts of the file, since the whole file needs to be compressed for each write and needs to be uncompressed for each read. Hence we choose to perform the compression and uncompression in units of logical blocks of the file.

To make data compression effective, the logical block size is selected as an integral multiple of the physical block size. The number of disk blocks required to store a logical block depends upon how well that block of data gets compressed. For example, if we choose the logical block size as 4K bytes and a logical block of data gets compressed to 2K bytes, then we need only 2 1024 byte sized physical disk blocks to store the 4K sized logical block.

The disk blocks corresponding to a logical block are stored at the corresponding index in the block entry table of the inode structure.

3.3 Buffering of Uncompressed and Compressed data

When *on-the-fly* compression and uncompression of data is incorporated into the file system, we need to know the exact size of the compressed data corresponding to the buffer being written, for the purpose of allocating required number of disk blocks for the corresponding logical block of the file. Since the device driver routines look at the size of the buffer cache to determine the number of sectors to be written

onto the disk, this information must be known before the corresponding buffer cache entry is placed on the disk queue.

In the design of our file system, the approach taken by the typical Unix-like file system, where data from the user buffer first gets copied into the buffer cache from where it is then written onto the disk, can still be followed in case of synchronous writes. Data from the user buffer is compressed into a temporary buffer, and the required number of disk blocks are then allocated for the logical block. Since the inode is kept locked till the write is finished, the corresponding block table entry is updated and the temporary buffer is placed on the disk queue for writes. The uncompressed data is also copied into a buffer cache entry as is done in a normal Unix-like file system.

In the case of asynchronous writes, the actual writing of data onto the disk takes place in background at regular intervals by the *bdflush* daemon. This daemon process periodically checks the buffer cache for entries which are marked dirty and flushes them onto the disk. Hence, to support the asynchronous writes in our file system, we either need to store the compressed data in the buffer cache or allow the compression to take place at the time the buffers are being flushed. The first alternative implies that uncompressed data can no longer be buffered, since the buffer cache can have only one entry for a logical block. This leads to paying a heavy penalty for file reads, since for every read, data from the buffer cache has to be uncompressed and given to the user. The disk block allocations cannot be delayed till the buffers are being flushed because these buffers in the first place cannot be put on proper queues as their size and associated disk blocks are not yet known. Thus they cannot be processed by the *bdflush* daemon.

One solution to this problem is to predict the number of blocks that would be required to store a logical block after it is compressed and allocate those many blocks initially. When the actual compressed size is known, either extra blocks need to be allocated or the surplus blocks need to be freed. This may lead to heavy disk fragmentation, which is undesirable.

3.3.1 Two Level Buffering

To overcome this problem in the design of our file system, we use the two level buffering mechanism, as implemented in Linux Version 1.3.51 onwards[3], to buffer the uncompressed and the compressed data. Buffering the uncompressed data retains the advantage of buffer cache, while buffering the compressed data is required to support the asynchronous writes.

The first level buffers are maintained as a set of virtual memory pages associated with a logical block of the file and store the uncompressed data. The buffer cache is used as the second level buffers which store the corresponding compressed data. Figure 3.1 depicts the scenario of the two level buffering.

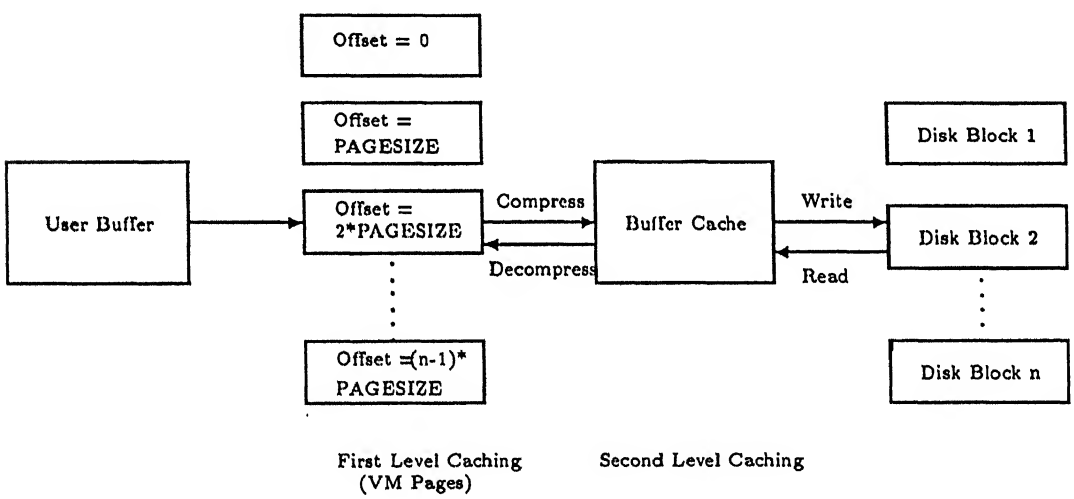


Figure 3.1: Buffering Mechanism

As shown in Figure 3.1, when a file is being written, data from the user buffer first gets copied into the corresponding memory pages in the first level buffer. With each set of these memory pages that correspond to a logical block of the file, there is an associated buffer cache entry, which forms the second level buffer. Data from the first level buffer is compressed and stored in the buffer cache, which gets written onto the disk, either synchronously or asynchronously by the *bdflush* daemon.

The disk block allocation for a buffer cache entry is done after the compressed data is written into it. This is because the actual number of disk blocks necessary for that buffer is known only after the data has been compressed.

Similarly when the file is being read, data from the disk is first read into the buffer cache. It is then uncompressed into the set of memory pages associated with that buffer cache entry. Any further reads on that block of data can thus be directly satisfied using these memory pages, without the need for uncompressing, or disk access.

Thus, while the memory pages maintain the advantage of speeding up the reads, the second level buffers aid in allowing writes to proceed asynchronously.

3.4 Support for Multiple Compression Techniques

As stated in the design objectives, the goal of our design is to make the process of compression and uncompression transparent to the user. At the same time, it also allows the user to choose the file system behavior with respect to the files he creates.

One goal of our file system is to support multiple compression techniques simultaneously in the file system. The choice of choosing a particular compression technique is available both to the system administrator and the user.

The system administrator can chose the compression technique to be used at the time of mounting the file system. Any file that will be newly created in this file system gets compressed by the compression algorithm chosen by the system administrator.

However, a user can override this default compression technique for the files he creates. He can decide whether the files that he creates be stored in the compressed form or not, and also can choose the compression technique that is to be used for the newly created files. New flags, that can be specified with the *open* system call have been added for this purpose. This option facilitates choosing an appropriate compression technique based on the file type and also the requirements of the user. Thus the compression technique used is an attribute of the file rather than the attribute of the file system, and this information hence is stored in the inode structure for that file.

Currently we have supported the Huffman Encoding[11], the 12-bit LZW compression technique[11] and the LZRW1 compression technique [14] in the implementation of our file system. Any new compression technique can also be easily added.

Chapter 4

An Implementation of the Compressed File System on Linux

We have implemented the compressed file system on Linux[8]. In this chapter, we start with a brief introduction to Linux and its file systems. We then explain the features of the Ext2 file system[2] of Linux, which is the base file system for our implementation. This is followed by a detailed discussion of the issues specific to our implementation.

4.1 Overview of Linux File Systems

Linux is a Unix-like operating system that was first developed for Intel 80x86 platforms and was later ported to other platforms. It was first implemented as an extension to the Minix Operating System[13]. Many modules of the kernel were later recoded and newer versions were released with advanced features.

In the early stages of development, the Linux file system was developed on the lines of the Minix file system. Later, to overcome the deficiencies of the Minix file system, two other file systems, the Extended file system(Extfs)[8] and the Xia file system (Xiafs)[8] were developed. To further overcome the problems in the Extfs, the Second Extended file system (Ext2fs)[2] was developed in 1993.

4.1.1 The Ext2 File System

- *Features*

The Ext2 file system provides many features common to a Unix file system, such as support for regular files, directories, device specific files and symbolic links. It provides support for long file names (up to 255 characters) by allowing varying length directory entries. Ext2fs also allows the system administrator to select the logical block size at the time of creating the file system, which will, in general be, an integral multiple of the physical block size. Using a larger logical block size can considerably speedup up disk access since fewer I/O operations are performed. However larger block sizes may lead to wastage of space in the last block of each file.

- *Physical Structure*

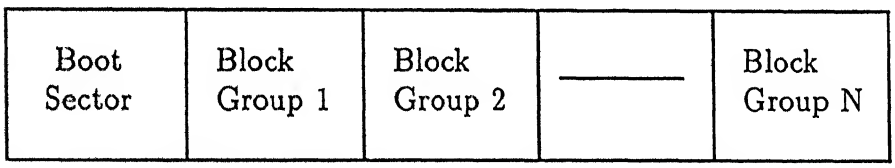


Figure 4.1: Structure of a Ext2 file system

The Ext2fs is structured as shown in Figure 4.1. The disk is divided into several block groups similar to the layout of the BSD file system[9]. Each block group contains a redundant copy of the super block and other file system control information, the block and inode bitmaps, the inode table and the data blocks for that group. Figure 4.2 depicts the structure of a block group.

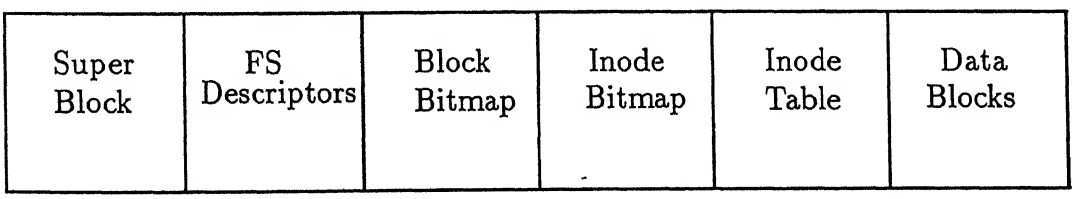


Figure 4.2: Structure of a Block group

The block bitmap is used to maintain the allocation status of the disk blocks in that group. Each bit in the bitmap corresponds to one disk block in the group. A value of 1 for a bit indicates that the corresponding block has been allocated and a value of 0 indicates that block has not yet been allocated. Similar is the case with the inode bitmaps.

The physical disk blocks allocated to a file are stored in a block table in the inode for that file. The first 12 entries of this table point to the direct blocks. The next three entries correspond to an indirect, double indirect and triple indirect blocks for that file. Figure 4.3 depicts the block table layout for a file in the Ext2 file system.

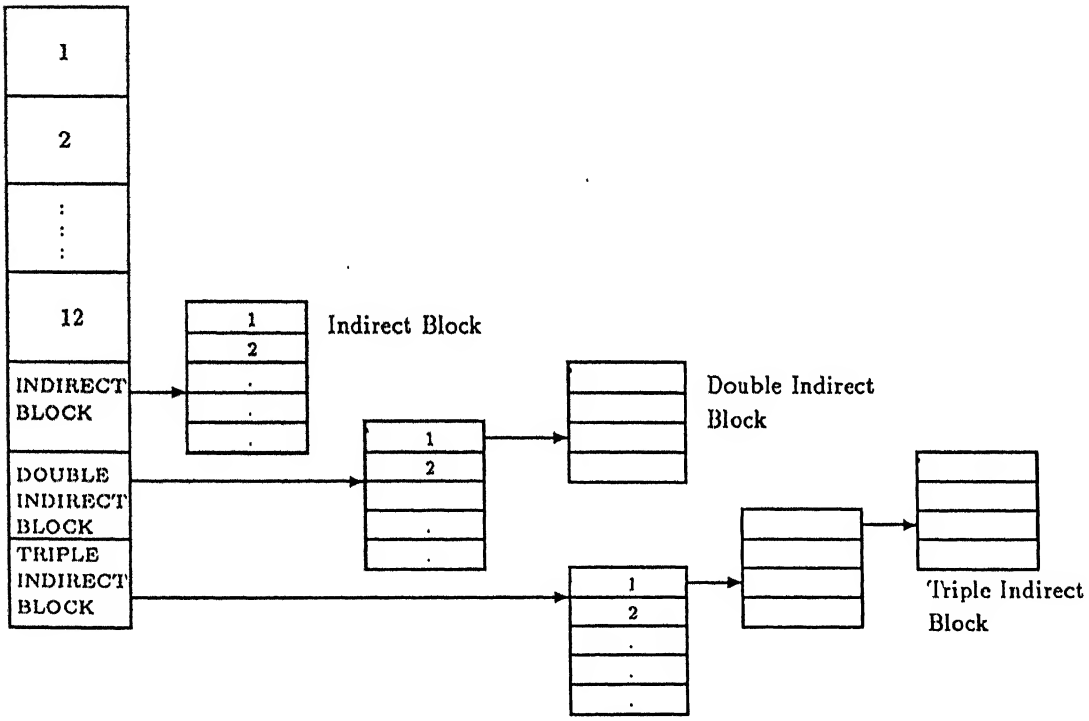


Figure 4.3: Block Table Layout for a file

Performance Optimizations

Ext2fs takes advantage of the buffer cache routines to perform read aheads in order to make reads faster. Further, Ext2fs has many allocation optimizations also. The block groups are used to cluster inodes and the corresponding data blocks. It also

ries to allocate the inode and data blocks for a directory and the files within it within a single block group, thus reducing the disk head seek times.

4.2 Issues in Implementation of the Compressed File System

4.2.1 Buffer Allocation Strategies for the Compressed file system

- *First Level Buffers*

As discussed in Chapter 3, the first level buffers are maintained as a set of virtual memory(VM) pages[1]. These VM pages are maintained as a hash table and are hashed upon the pair <inode,offset>, where *inode* is the inode number of the file and *offset* is the offset of the file data which is stored in this VM page. Further, all the VM pages that correspond to a file are linked in a doubly linked list. The VM pages that correspond to a buffer cache entry are identified by comparing the sizes of the VM page and that of the logical block size of the file system, and finding the first VM page corresponding to a buffer cache entry.

When a file is being written at an offset, data from the user buffer is first copied nto the corresponding VM pages. If the VM pages are not present, they are allocated nd the copying of data is done.

Second Level Buffers

ie second level buffers in the Linux kernel are the buffer cache entries. Each buffer che entry corresponds to one logical block of file data. When a file is being written, soon as the data corresponding to a logical block is written into the VM pages, compressed and stored in the corresponding buffer cache entry.

f the corresponding entry is not found in the buffer cache, a new buffer is created. entry is however not placed in the buffer cache queues, since no disk blocks associated with this entry. Data from the corresponding VM pages is then

compressed and stored in this buffer. Knowing the compressed size, we now allocate the required number of disk blocks and associate them with this buffer. The buffer is now inserted into appropriate buffer cache queues.

If the required entry is found in the buffer cache, but the new compressed size exceeds the size of the allocated blocks for that entry, a proper reallocation of blocks is done so that required number of contiguous blocks are allocated.

4.2.2 Disk Block Allocation and Representation

- *Allocation*

After the file data has been compressed into a buffer, we allocate the required number of disk blocks for that logical block of data. We then associate these disk blocks with the buffer and store them in the block entry table of the inode. For efficiency we allocate all the blocks for a logical block in a contiguous fashion on the disk. It then suffices that we store the first block number as the physical block number of the buffer. This along with the size of the buffer (the compressed size) determines the number of blocks that correspond to this logical block of data. Contiguous allocation of disk blocks for a logical block helps to get good performance by allowing the kernel to read a logical block with a single disk transfer.

The disk is partitioned into multiple block groups in a way that is done in Ext2fs. A block bitmap is used to represent the availability of disk blocks within a block group. A bit value of 0 indicates that the corresponding block is free and a value of 1 indicates that the block is allocated.

We have used the goal block allocation policy similar to that used by the Ext2fs. The goal block allocation policy tries to allocate contiguous blocks on the disk to store a file. For allocating disk blocks for a logical block of the file, the block following the last allocated block for the previous logical block of the file is designated as the goal block.

The algorithm for allocating disk blocks first searches for the required number of free blocks starting from the goal block. If the required number of free blocks are found starting at the goal block, the search ends.

In case the required number of free blocks are not found at the goal block, we search in the near vicinity of the goal block to satisfy the request. This search is limited to the next 64 bits of this block group.

If the request is not yet satisfied, we then search for a free byte in this block bitmap. Starting from the first bit of this free byte, a search is made backwards to locate the first 0 bit in this sequence of 0 bits. Starting from this first free bit, we allocate the required number of blocks. Searching for a free byte has the advantage that block allocation request for the next logical block of the file can be satisfied at the goal block.

If a free byte is not found, the entire block bitmap is scanned for the required number of free blocks. If this too fails, then each of the remaining block groups is tried until we get the required number of contiguous free blocks.

Representation

The other issue we need to deal with is the way the disk blocks are represented in the block entry table of the inode. In Ext2fs, a 32 bit number is used to represent a physical disk block. For the compressed file system however, each logical block corresponds to multiple (consecutive) physical blocks.We represent this information in the block table in the following way.

Of the 32 bits that are used to store the physical block number, we allocate n bits to store number of blocks and the rest of $32 - n$ bits to store the first block number. The value of n depends on the logical block size chosen. If we choose a logical block size of L bytes and the physical block size is P bytes, then the maximum number of physical blocks needed to store L bytes of data is $N = L/P$. Thus we need at most $\log_2(N)$ bits to store the number of physical blocks required. As an example, if the logical block size is 8K bytes and the physical block size is 1K byte, then we need at most $\log_2(8) = 3$ bits to store the number of blocks required. Thus the remaining 29 bits can be used to store the first block number, which allows to represent 2^{29} disk blocks. This implies that the maximum total file system size is less than that of Ext2fs. But in practice however, since these numbers are enough to represent a very large file system, using 29 bits instead of 32 bits to represent the block number

COMPR_ALGO = none	No compression is done by default for any newly created files
COMPR_ALGO = default	The default compression technique, currently the LZRW1 compression
COMPR_ALGO = lzrw1	The LZRW1 compression technique
COMPR_ALGO = lzw12	The 12-bit LZW12 compression technique
COMPR_ALGO = huffman	The Huffman encoding technique

Table 4.1: IDs for Compression Techniques Supported

does not make much of a difference.

4.3 Implementation of System Calls

As explained earlier, file system related system calls in Linux have a file system independent portion and a file system specific portion of the code. In this section we present the details of implementation of the file system specific portion of the code for the *mount*, *open*, *read* and *write* system calls.

4.3.1 Mounting a file system

The file system specific portion of the code for mounting a file system involves processing any file system specific options to the mount call and then reading the super block of the file system being mounted. As explained in Chapter 3, the compression technique that will be used as a default for any newly created files in the compressed file system has to be specified at the time of mounting the file system. We use the file system specific mount options to specify this default compression technique.

A new <string=value> pair is added to the mount options for this purpose. In order to specify the compression technique to be used, we need to pass the string compress=COMPR_ALGO as a mount time option. The value of COMPR_ALGO is interpreted as shown in Table 4.1.

Based upon the value of COMPR_ALGO, a global variable *filesystem_compressid* is set to the corresponding constant that identifies the specified compression technique.

The value of this global variable is assigned to the *i_compressid* field of the inode structure for any newly created files in the file system. This value of this field is thus used to determine the compression and uncompression technique to be used with this file.

4.3.2 Opening a new file

The file system specific portion of the *open* system call is used to choose a compression technique of choice for the file being created. In this way, the default compression technique specified with the *mount* call can be overridden, thus allowing multiple compression techniques to be used simultaneously in the file system. Similar to the mount option, the compression technique to be used for the file being created can be specified as a flag with the *open* system call. The *i_compressid* field of the inode structure for this file will now be assigned the ID of the compression technique thus specified. The following is an example of the *open* system call for overriding the default compression technique chosen for the file system.

```
fd = open("/tmp/example", O_CREAT|O_HUFFMAN_COMPR, 0644);
```

In the above example, `O_HUFFMAN_COMPR` could have been replaced with `O_LZRW1_COMPR` for LZRW1 compression or with `O_NOCOMPR` for no compression.

4.3.3 Reading from a file

As explained in Chapter 3, the read system call in Linux gets directed to the *generic_file_read* routine, which is a file system independent interface for reading the data from the VM pages associated with the file into the user buffer. The process of updating the VM pages with data from the buffer cache is implemented as a file system specific routine. In this subsection, we trace the *generic_file_read* routine and explain the file system specific *readpage* routine, which is used to update the VM pages.

Using the inode of the file to be read and the offset from which file read is to be initiated, the hash table of the VM pages is looked into, for the corresponding VM

page. If the VM page is present and is up-to-date, data starting from the specified offset is copied into the user buffer.

If the corresponding VM page is not present, a new page is created and its fields are properly initialized. If either the page is newly created or the page exists and is not up-to-date, the file system specific routine to update the page is called.

The *readpage* routine of the compressed file system performs the following actions.

1. Given the offset from which the file read is to be initiated, the logical block corresponding to that offset is calculated. Using the block table entries in the inode structure, the first physical block corresponding to this logical block is obtained.

2. If the appropriate buffer cache entry is found and is up-to-date, data from the buffer cache entry is transferred into the VM pages associated with this buffer. For uncompressed files this implies a simple copy, while for compressed files this implies uncompressing the data from the buffer cache into VM pages. The exact size of the data that is to be uncompressed is available in the first two bytes of the compressed data. The number of VM pages associated with a buffer cache entry is a function of the page size of the system and the logical block size of the file system. Data from these updated VM pages is then copied into the user buffer.

3. If the corresponding entry is not found in the buffer cache, data is to be read from the disk. For this purpose, an entry is created in the buffer cache and data from the disk is transferred into this buffer. For compressed files, this data is then uncompressed into the corresponding VM pages, which is then copied into the user buffer. For uncompressed files, a direct copy from buffer cache to VM pages suffices.

In the implementation of the compressed file system, we have also provided an option for reading the raw compressed data from the file. If the `O_RAWREAD` flag is specified when the file is opened, any reads on that file descriptor return the data

in the compressed form. For such reads, we bypass the *generic_file_read* routine. Instead of uncompressing the data from the corresponding buffer cache entry into the VM pages from where it is then copied into the user buffer, the compressed data from the buffer cache is directly copied into the user buffer.

As can be observed, file data needs to be uncompressed only when it is not present in the VM pages. Once the VM pages contain the data in the uncompressed form, any further reads of the same data does not involve either disk access or the overhead of uncompression.

4.3.4 Writing to a file

In this subsection, we explain the file system specific portion of the write system call. We present the implementation details of write, both for a file which is stored in the compressed form and for one which is stored in uncompressed form.

- *Writing to an uncompressed file*

From the offset at which the write is to be initiated the logical block corresponding to that offset is calculated. The block entry table of the corresponding inode is used to obtain the physical blocks that correspond to this logical block.

If the entry in the block entry table indicates that no physical blocks are yet allocated for this logical block, the required number of physical blocks are then allocated using the proper block allocation strategy.

The buffer cache entry corresponding to this logical block is then obtained. If the entry is not up-to-date and only a part of the logical block is being written, a disk read is initiated for this buffer.

Data is then copied from the user buffer to the buffer cache entry at the proper offset. Data is also copied into the corresponding VM pages, if they exist.

If the file is to be written synchronously, the corresponding buffers are placed on the disk queue. In case of asynchronous writes, the *bdflush* daemon takes care of writing the data onto the disk. This is exactly the same as for Ext2fs.

- *Writing to a compressed file*

Writing to a file which is stored in the compressed form differs from writing to an uncompressed file in the way disk blocks and the corresponding buffer cache entries are allocated for the file. In case of writing to an uncompressed file, we know before hand exactly how many disk blocks are needed for a logical block. This is known from the physical and logical block sizes of the file system. On the contrary, for a file that is to be stored in compressed form, this information becomes available only after the compressed size of the data corresponding to that logical block is known. This implies that allocation of disk blocks to a logical block of the file can be done only after the logical block data is compressed. This needs the following processing to be done for writing a logical block of the file.

After finding the logical block to be written starting at the current file offset, the block entry table of the corresponding inode is used to locate the physical disk blocks for this logical block. If no blocks are yet allocated, a stand alone buffer is created for this logical block. This buffer does not appear in any of the buffer cache queues. Data from the user buffer is first copied into the corresponding VM pages. The data from these VM pages is then compressed and written into this buffer. Since the size of the compressed data need not always be an integral multiple of the physical block size, we need to know the exact size of the compressed data while uncompressing it for reads. Hence this size is stored in the first two bytes of the compressed data. The size of the buffer is changed to reflect the compressed size. Since buffer sizes should only be in multiples of physical block sizes for the purpose of disk accesses, the buffer size is changed to the integral multiple of the physical block size just greater than the compressed size.

After thus knowing the size of the buffer, we now know the number of physical blocks required to store this logical block on the disk. The block allocation routine is used to allocate the required number of physical blocks for this logical block. This information is stored in the buffer and at the corresponding index of the block entry table of the inode. The buffer is then placed on the appropriate buffer cache queues.

On the other hand, if the disk blocks for this logical block have already been allocated, the buffer cache is checked to find the corresponding entry. If an entry is

found, it is updated by the data from the corresponding VM pages. The updated buffer size is then checked against the available size for this logical block on the disk. If the number of disk blocks allocated for this logical block are such that the updated buffer does not fit into them, a reallocation of disk blocks for this logical block needs to be done. For this purpose, we first check if the extra blocks can be allocated contiguous to the disk blocks currently allocated. If it cannot be done, then the buffer is first removed from the queues. Previously allocated disk blocks are then freed and required number of disk blocks are allocated again. The buffer and the corresponding block entry table of the inode are updated accordingly, and the buffer is again inserted into the buffer cache queues.

4.4 Avoiding Compression of Small Files

One problem that is encountered in a compressed file system is regarding the storage of smaller files. Storing smaller files in compressed form does not give any effective advantage, since both the compressed and uncompressed data require almost the same number of blocks in many cases. Moreover, we need to pay the penalty for compressing and uncompressing the file for writes and reads. In this section, we present an approach that we have followed in the implementation of our compressed file system to solve this problem. This approach allows us to store the smaller files in uncompressed form and compress only those files that are larger than a threshold.

We define a threshold size for the file system. All the files whose size is less than the threshold are stored in uncompressed form. Once the file size crosses the threshold, we start storing the file in the compressed form. Our technique requires this threshold size to be less than the logical block size of the file system.

When the file is first created, the *i_comprmode* field of the inode is set to FLEX_FILE_UNCOMPRESSED. This implies that the file will be stored in the uncompressed form. The *i_comprmode* flag is checked during the time of reading the first logical block of the file. If it is set to FLEX_FILE_UNCOMPRESSED, data from the corresponding buffer cache entry is copied into the VM pages without uncompression.

Now, when the size of the file exceeds the threshold value and the O_NOCOMPR

flag was not specified while opening the file, the *i_comprmode* field is changed to FLEX_FILE_COMPRESSED. This implies that the file will now have to be stored in the compressed form. Since we require the threshold value to be less than the logical block size, the change from uncompressed mode to the compressed mode always takes place only while writing the first logical block of the file. This implies that no extra reads or writes are required when the storage mode gets changed from FLEX_FILE_UNCOMPRESSED to FLEX_FILE_COMPRESSED. After the data is copied from the user buffer into the corresponding buffer cache entry, the following processing has to be done when the mode of storage has changed from FLEX_FILE_UNCOMPRESSED to FLEX_FILE_COMPRESSED.

1. Data from the buffer is compressed into a temporary buffer and this compressed data is then copied back into the buffer.

2. The size of the buffer is changed so as to reflect the size after compression.

3. To make the buffer copy and the corresponding disk copy of the data consistent, the buffer is written synchronously onto the disk. This also takes care of the situation when the *readpage* routine, which reads data from the buffer into the VM pages, cannot locate this buffer in the buffer cache because of the change in the size of the buffer.

The *readpage* routine is also modified to handle the situation when the storage of the file changes from FLEX_FILE_UNCOMPRESSED to FLEX_FILE_COMPRESSED.

When copying the data from the buffer that corresponds to the first logical block of the file, the *readpage* routine now checks for the storage mode of the file. If it is FLEX_FILE_UNCOMPRESSED, a direct copy is made, else the data is uncompressed and the uncompressed data is copied into the VM pages.

4.5 Support Utilities

Support utilities have been developed for Ext2fs to create a file system, modify and correct any inconsistencies in the file system etc. Since we took Ext2fs as the base

file system for our implementation, we took the Ext2fs code for these utilities and made minor modifications to it in such a way that they work for our file system. In this section, we mention the utilities that have been provided, and the modifications that we made to suit them for our file system. Detailed explanation about each of these utilities is available in [2].

- **mkfs**

  The *mkfs* utility is used to create an empty file system on a disk partition. Options for selecting the block size, bytes/inode ratio, percentage of reserved blocks for the super user etc., can be specified with *mkfs* for creating a file system with the desired parameters.

  For creating a compressed file system, we have added the following, to the options that can be specified with *mkfs*.

  - An option for specifying the size of the logical block, in bytes. If this option is not specified, a default value of 4K is taken as the logical block size.
  - An option for specifying the threshold size for files, below which all the files are stored in uncompressed form. This option has to be specified as a fraction of the logical block size. A default value of 0.5 is taken, if this option is left unspecified.

- **tunefs**

  The *tunefs* utility is used to modify the file system parameters such as the maximal mount counts between two file system checks, the percentage of the reserved blocks for super user etc. We have adopted this code from Ext2fs, without any modifications.

- **fsck**

  The *fsck* utility is used to detect and repair any inconsistencies in the file system. *Fsck* for Ext2 runs in several passes, each pass checking one consistency feature of the file system.

The routine that reads the physical block numbers from the block entry table of an inode has been modified to reflect the change in representation of the disk blocks as explained in subsection 4.2.2. We have also added the consistency check to verify whether the number of physical blocks that correspond to a logical block of the file is in accordance with the logical block size of the file system.

4.6 Compression Techniques Used

In the implementation of the compressed file system, we have included the Huffman encoding[11], which uses a statistical model, the 12-bit LZW compression technique[11], and the LZRW1 compression technique[14], both of which use the dictionary based schemes. In this section, we present the details of the Huffman encoding technique and the LZRW1 compression technique. Details of the 12-bit LZW compression technique, which varies from the LZRW1 technique only in the way the dictionary is maintained, are available in [11].

4.6.1 Huffman Encoding Scheme

The Huffman encoding scheme[11] generates variable length codes for symbols. A code contains integral number of bits. Symbols with higher probabilities are encoded using shorter codes and symbols with lower probabilities get larger codes. The unique prefix attribute of the Huffman codes allow them to be correctly decoded despite being of variable length. Decoding a stream of Huffman codes is generally done by following a binary decoder tree.

- *Procedure to Build the Code Tree*

The procedure for building the Huffman code tree is simple. The tree is built in the following way.

1. The symbols are laid out as leaf nodes, each with a weight equal to the frequency or probability of its occurrence. All the nodes are marked free.

2. The two free nodes with the lowest weights are located. A parent node for these two nodes is created, and is assigned a weight equal to the sum of the two child nodes.

3. The parent is added to the list of free nodes and the two child nodes are removed from the list.

4. One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit, and the other when decoding a 1 bit.

5. Steps from 2 through 4 are repeated until only one free node is left. This free node is designated as the root.

Decoding the Huffman encoded stream requires a simple tree traversal starting from the root, and moving down either to the left child or to the right child depending upon the next bit in the code. The symbol corresponding to the leaf node reached is the required decoded symbol. The implementation details for Huffman encoding are available in [11].

4.6.2 LZRW1 Compression Algorithm

The LZRW1 compression algorithm is based upon the adaptive dictionary based compression technique proposed by Lampel-Ziv[11], which was explained in Chapter 2. The LZRW1 compression technique uses a simple hash table as a dictionary, which makes the construction and maintenance of the dictionary quite simple, thus making this compression algorithm strike a balance between the amount of compression achieved and the time taken for compression and uncompression. Figure 4.4 depicts the LZRW1 compression algorithm.

- *LZRW1 Compression Algorithm*

1. The LZRW1 algorithm uses a single pass literal/copy mechanism on the input text. Each time, the next few bytes of the input are either transmitted directly to the output, or as a pointer to the already processed stream of the input.

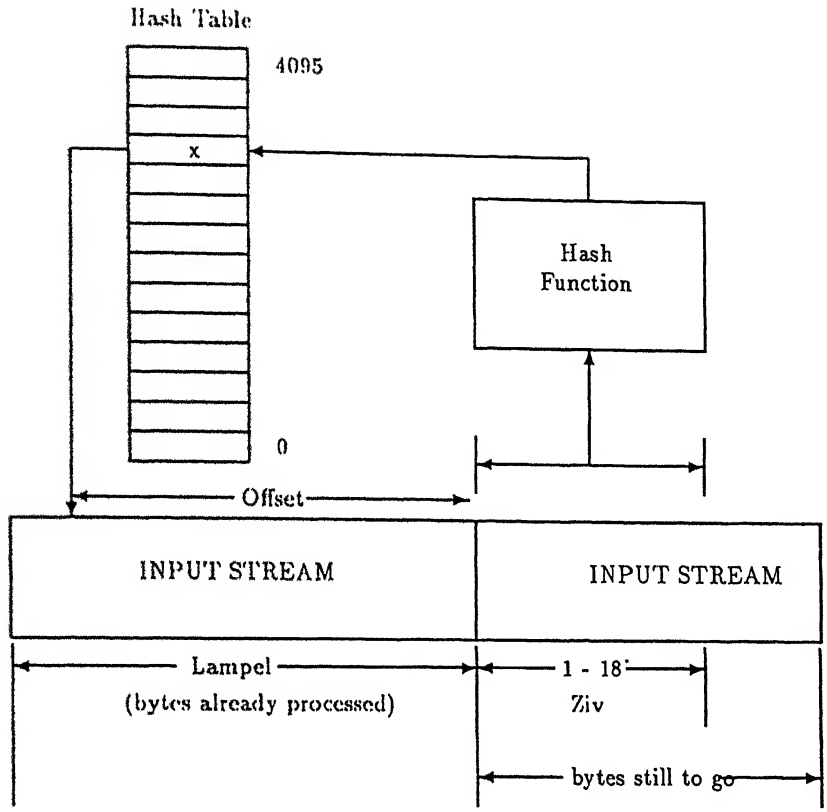


Figure 4.4: LZRW1 Compression Technique

2. At each step, the next three bytes of the input are used to hash into the hash table to find a match into the previously processed input stream. If a match is found, the <offset,length> pair is transmitted to the output and the hash table entry is updated to point to the first byte of currently examined three bytes. The input pointer is advanced by the length of the match.

   The offset in the <offset, length> pair indicates the relative position backwards, of the starting of the matched string from the current input pointer. The length indicates the length of the match found.

3. If the hashed entry does not point to a valid offset of the input text, the first of the three bytes is transmitted as a literal item, the input pointer is advanced by 1 and the process is continued.

4. A control bit is maintained for each item transmitted to the output. The corresponding control bit is set as 0 for a literal item and 1 for a matched item.

LZRW1 Uncompression Algorithm

The uncompression algorithm is extremely simple and fast. It starts by looking at the control bits and thus deciding upon the next item of the input stream (the compressed data). If the next control bit seen by the uncompresser is 0, it knows that the item following is a literal, and hence directly copies the next byte from the input to the output. On the other hand, if the next control bit is 1, the uncompresser treats the next two bytes as an <offset,length> pair, calculates the offset and the length from them, and copies the corresponding number of bytes starting at that offset, from the previously uncompressed text to the current position of the output stream. The input and output streams are then advanced correspondingly.

This approach makes uncompression a fairly simple job, since it need not maintain any hash table information and a single pass over the input stream is enough to carry out the uncompression. This property makes the algorithm particularly suited for our needs since reads typically constitute a large majority of the file system operations and have to be consequently faster.

Implementation Details

The LZRW1 compression algorithm uses the hash function in accordance with the advice in[4] which seems to be an optimal one. We take a hash table of 4K size and use the next three bytes of the input to hash into this table using the hash function

$HASH(ptr) = (((40543 * (((*(PTR)) << 8) \wedge ((*((PTR) + 1)) << 4) \wedge (*((PTR) + 2)))) >> 4)\&0xFFF)$

A literal item is transmitted as it is to the output stream, whereas, a copy item is transmitted as a two byte <offset,length> pair to the output. Of the two bytes used for this purpose, we use 12 bits to store the offset into the previously seen input

text, and 4 bits to store the length of the match. This restricts the dictionary to a size of 4K bytes and the length of a match to at most 16 bytes.

Since we thus require two bytes to transmit a copy item, we look for a match of at least three bytes, in order to make the compression effective. Thus we transmit a copy item only if the length of the match is greater than or equal to 3 bytes. Taking this into consideration, and the fact that we use 4 bits for storing the length information, we can actually extend the length of the match to 18 bytes by considering the length bit patterns 0000 through 1111 as matches of lengths 3 through 18.

Because the algorithm checks all the pointers that it fetches from the hash table, the hash table need not be initialized. The LZRW1 algorithm updates its hash table once after every item rather than every byte, making the hash table update rate inversely proportional to compression.

Chapter 5

Performance Measurements

The performance of the compressed file system is affected by two factors, the time lost in compression and uncompression of file data, and the time gained because of lesser data transferred to/from the disk. Thus, by using a good compression technique which is fairly fast, the time lost in compression and uncompression can be made up to some extent by the time gained because of lesser disk transfer required. Hence we hypothesized that the performance of our compressed file system would not be much worse than that of Ext2fs.

In order to validate this hypothesis, we ran benchmark programs and compared the performance of our file system with that of the Ext2fs. In this chapter, we describe the experiments we have conducted, explain the parameters that have been measured, present the results, and analyze them to evaluate the performance of our file system.

We used a Pentium 133 MHz processor with 32 MB of main memory to run the benchmarks. The hard disk had the following characteristics.

Number of cylinders = 4385

Number of sectors per cylinder = 63

Number of heads = 16

Disk head RPM = 5400

5.1 Timing statistics for transfer of a logical block

We conducted experiments to measure the time required for reading and writing one logical block of data. We compared the statistics obtained for our file system with that of an Ext2 file system with the same logical block size.

For this purpose we created a compressed file system with the logical block size equal to 4K bytes. We also created an Ext2 file system with the same logical block size. We chose a mix of file types (C source files, binary executables, text files etc.,) with an aim of achieving reasonable averages for the compression ratio. We used the LZRW1 compression technique for compression and uncompression of file data.

5.1.1 Parameters Measured

We have measured the following parameters while reading and writing the files in our file system. These parameters are also measured for the Ext2 file system. All the values are the time taken for reading/writing a single logical block (4K). For each of these parameters, we have measured both the CPU times and the real times.

- *Time for Synchronous Writes*

The CPU time for synchronous writes includes time taken for allocation of disk blocks and buffer cache entries, time taken for copying data from the user buffer into buffer cache entry and the associated VM pages, and the time required for inserting the buffer cache entry into the disk queues for writing onto the disk. For the compressed file system, time taken for compression is also included in the CPU time.

The real time is the total time spent in the system call. Apart from the CPU time, it includes the time taken for transferring the data from the buffer cache entry onto the disk.

In order to measure these timings, we created files of different types (C sources, binary executables, text files etc.,) in the compressed file system and in the Ext2fs. We restricted the size of these files to 48K inorder to avoid the overhead of writing the indirect blocks. Then, we measured the times by over writing these files in

synchronous mode and taking the average of the obtained timings. Measuring the times while overwriting the files is to avoid the overhead of writing the block bitmap and the inode bitmap onto the disk.

Time for Asynchronous Writes

The CPU time for asynchronous writes will be the same as that of the synchronous writes, excluding the time required for inserting the buffers into disk queues. This will be done by the *bdflush* daemon in the background.

Real time for asynchronous writes does not include the disk transfer time since it will be done in the background by the *bdflush* daemon. Hence this time will be much lesser than the real time for synchronous writes.

In order to measure these timings, the same setup as used for the synchronous writes was used, except that thr files are written in asynchrnous mode.

Time for Unbuffered Reads

The CPU time for unbuffered reads include the time required to allocate virtual memory pages for the logical block being read, creating buffer heads for these VM pages, time taken for inserting these buffer heads into the disk queues and the time taken for transferring data from the VM pages into the user buffer after data from the disk has been read into these VM pages. In case of the compressed file system, the time required for uncompressing the data while transferring it from the disk also gets included.

Real time for unbuffered reads is the total time spent in the read system call. It includes the CPU time, the time taken for transferring the data from the disk into the corresponding VM pages, and the time taken because of context switches etc.

In order to measure these timings, we created several files of size 4K in the compressed file system and in the Ext2fs. We then rebooted the system in order to ensure that the data of these files is not in the buffer cache. Now, each file was read once and averages of the measured times was taken.

	Synchronous Writes		Asynchronous Writes	
	CPU Time	Real Time	CPU Time	Real Time
Compressed fs	296 μsecs	6852 μsecs	280 μsecs	1273 μsecs
Ext2fs	131 μsecs	6692 μsecs	122 μsecs	169 μsecs
Ratio of Comprfs to Ext2fs	2.25	1.02	2.29	7.53

Table 5.1: Times for Writes

	Unbuffered Reads		Buffered Reads	
	CPU Time	Real Time	CPU Time	Real Time
Compressed fs	102 μsecs	2106 μsecs	10 μsecs	142 μsecs
Ext2fs	45 μsecs	2086 μsecs	10 μsecs	141 μsecs
Ratio of Comprfs to Ext2fs	2.26	1.01	1.00	1.00

Table 5.2: Times for Reads

- *Time for Buffered Reads*

Since buffered reads return data directly from the VM pages, CPU time for buffered reads include only the time required for transferring the data from VM pages into the user buffer. This time should be the same for both the Ext2fs and the compressed file system.

Real time for buffered reads is the total time spent in the execution of the read system call. Since it does not include the disk transfer time, real time for buffered reads will be much lesser than that of the buffered reads.

In order to measure these times, we first read all the files to ensure that they are now in the buffer cache. The same files were read multiple number of times and the averages of the measured times was taken.

5.1.2 Results and Analysis

Table 5.1 presents the average CPU times and real times required for writing one logical block of data to the compressed file system and to the Ext2fs. In case of

Time for Compression	Time for uncompression
132 μsecs	51 μsecs

Table 5.3: Times for Compression and Uncompression

synchronous writes, the CPU time required for writing one logical block of data into the compressed file system is more than twice the time that is required for a similar write into the Ext2 file system. This is because of the extra time spent in compressing the data before writing it onto the disk. The ratio of the real times however indicate that the time lost in compression of file data is compensated to some extent by the time gained because of lesser disk transfer involved.

In case of asynchronous writes, the CPU times and the real times indicate that the compressed file system performs badly in comparison with the Ext2fs. Since disk transfer in this case takes place in the background by the *bdflush* daemon, and the compression takes place in the foreground, the time lost in compression cannot be compensated for by the time gained by lesser disk transfer.

Table 5.2 presents the average CPU times and real times required for reading one logical block (4K) of data from the compressed file system and the Ext2fs. As seen, the CPU time required for an unbuffered read of 4K data from the compressed file system is over two times that required for a similar read from the Ext2fs. This is clearly because of the extra time spent in uncompressing the data. However, the ratio of real times for both the file systems show that the time lost in uncompression is almost compensated for by the time gained because of lesser disk transfer involved.

In case of buffered reads, since the uncompressed data is already available in the virtual memory pages, the CPU times and the real times for the compressed file system are equal to that of the Ext2fs.

Table 5.3 presents average CPU times taken for compressing and uncompressing one logical block (4K) of data using the LZRW1 compression technique. We chose a mix of file types (C source files, binary executables, text files etc.,), compressed the file data in chunks of 4K and uncompressed it back, to obtain the average values for compression and uncompression.

From the results presented above, we see that in case of reads the performance of the compressed file system is almost as good as that of the Ext2fs. While synchronous writes are also as fast as that of Ext2fs, the performance loss is significant in case of asynchronous writes. We are at a loss to explain this degradation of performance in case of asynchronous writes.

Since reads typically constitute a large majority of the file system operations, and since most of the read requests are satisfied by the data readily available in the virtual memory pages, the overall performance of the compressed file system is not much worse as compared to that of the Ext2 file system.

Chapter 6

Conclusions

6.1 Summary

In this report, we have discussed the design and implementation of a file system for UNIX, with the feature of *on-the-fly* compression and uncompression of file data in a way that is transparent to the user.

We have implemented our file system on the Linux operating system. Issues like allocating the disk blocks to a logical block of the file, representing the allocated blocks in the block entry table of the inode corresponding to that file, options for specifying the compression techniques to be used etc., which are specific to the implementation, have been discussed in detail.

We started with the hypothesis that by using an efficient compression technique which is fairly fast, the extra time spent in compressing and uncompressing the file data is compensated to some extent by the time gained because of lesser disk access. In order to validate our hypothesis, we conducted experiments to evaluate the performance of our file system and compared the results with that of the Ext2fs. From the results of these experiments we observe that the performance of the compressed file system is almost as good as that of the Ext2fs in case of reads and synchronous writes. Though there is a loss of performance in case of asynchronous writes, since reads typically constitute a large majority of the file system operations, the overall performance of our file system is not much worse when compared with

Ext2fs.

Hence, though disk space is not at premium these days, going for a compressed file system to get the advantage of increased effective disk space at a little cost in terms of extra time may not be a bad idea.

6.2 Limitations

Our implementation currently has the limitation that the system cannot boot from the compressed file system. This is because *lilo*, the program which installs the boot loader is unaware of the fact that files in our file system are stored in the compressed form. Hence it cannot read the disk blocks corresponding the kernel image when the system is rebooted. In order to over come this limitation, the code for *lilo* has to be changed in such a way that apart from storing the disk blocks corresponding to the kernel image, it also stores the information about the uncompression technique that is to be used while reading these disk blocks. Apart from this, the *on-the-fly* compression feature of our file system is totally transparent in all other ways.

References

[1] M Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[2] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and implementation of the second extended file system. *Proceedings of the First Dutch International Symposium on Linux*, 1990.

[3] Michael Elizabeth Chastain. Kernel change summaries for linux releases. ftp://ftp.shout.net/pub/users/mec/kcs/v2.0/.

[4] D.E.Knuth. *The Art of Computer Programming*, volume 23. Addison-Wesley Publishing Company, Reading, Massachusetts, 1973.

[5] HewlettPacard. Doublespace. http://hpcc920.external.hp.com/isgsupport/cms/docs/lpg

[6] IBM. Superstor. http://ourworld.compuserve.com/homepages/rpr/SuperSTO.HTM.

[7] IBM. Superstor. http://www.bergen.org/ edwdig/geos/geoinfo/superstor.html.

[8] Michael K Johnson. Linux kernel hackers' guide, version 0.7. ftp://sunsite.unc.edu/pub/Linux.

[9] Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley Publishing Company, 1989.

[10] Macintosh. Doumentation on stacker. http://www.stac.com/pss/techmac.html.

[11] Mark Nelson. *The Data Compression Book.* M&T Books, New York, 1992.

[12] Sandberg.R, D.Goldberg, S.Kleiman, D.Walsh, and B.Lyon. Design and implementation of the sun network filesystem. *Proceedings of the USENIX Conference*, pages 119–131, Summer 1985.

[13] A Tanenbaum. *Operating Systems: Design and Implementation.* Prentice Hall India, 1987.

[14] Ross N Williams. An extremely fast ziv-lampel data compression algorithm. *IEEE Computer Society Data Compression Conference*, pages 8–11, April 1991.

Date Slip

This book is to be returned on the date last stamped.

A 123643

CSE-1997-M-PRA-DES